COSC329 Assignment on Combinatorial Generation

Implement the following algorithms for permutation generation in C and confirm the outputs are correct for n=4. Then remove the output statement and measure the computing time for each method with n=10, 11, and 12, or 11, 12, and 13, depending on the speed of your computer. If the computer warrants, you can choose a larger n. Choose (6) if your student number is even, or (7) if odd. You can do both.

(1) Algorithm 5  
(2) Algorithm 6  
(3) Algorithm 7  
(4) Algorithm 8  
(5) Algorithm 9  
(6) Meta algorithm that generates a nested loop structure such as one in page 9. In a meta algorithm, a parent program and a child program perform combinatorial generation co-operatively. See attached for a sample program.  
(7) Enhanced version of any of the above that can test a sorting program. You should make your own sorting program, such as quicksort, heapsort, and mergesort.

Present the following in hardcopy

Cover page: Assignment title, name, student number, student account
A. Source lists and documentation for (1) – (6) or (7). A sample documentation, a meta/generated program, and a tester/tested program are attached.
B. Outputs for n=4 for (1) – (6) or (7). For (7) see attached.
C. Time measurements with the above mentioned values of n for all of the above six methods (1) –(6) or (7). See page 4 for (7)
D. Discussions on the performances of those methods and possible suggestions for improvements

Note. For the meta algorithm approach, you need to present the generated program as well as the generating program. The time must include the compile time for the generated program. See the attached sample program for the meta program approach. Items A-C are rather straightforward. Item D will make a difference in the grade.

Due: 5:00 pm, 31 May 2007. Drop due: one week later.
Worth 20%

In addition, you are required to submit electronically prog1.c, …, prog5.c, corresponding to the above (1), …, (5) and meta.c for (6) or tester.c and sort.c for (7). For electronic submission, follow the guideline below.

These programs must be able to be run with “gcc” and “a.out”. The user is supposed to give only n. For (6), only a meta program will do as the child program is generated from it. For (7), follow the attached example.

Grading policy. A = 40, B = 10, C = 10, D = 40 in percentage
Documentation of Algorithm 5

\begin{verbatim}
var used: array[1..100] of Boolean;
var a: array[1..100] of integer;
procedure perm(k);
  begin
    if k<=n then begin
      for i:=1 to n do
        if not used[i] then begin
          a[k]:=i;
          used[i]:=true;
          perm(k+1);
        end
    end;
    else output(a)
  end;
begin {main program}
  for i:=1 to n do used[i]:=false;
  perm(1)
end.
\end{verbatim}

Variables: \( n \): size of permutations
\( i \): item \( i \)

Procedures: perm: the main recursive procedure with parameter \( k \)

Arrays: a: container of permutations,
used: Boolean array to indicate if item \( i \) is used or not.

This algorithm generates permutations in lexicographic order. The procedure “perm” is a recursive one. The parameter \( k \) controls the position of the container array “a”. In “perm”, item \( i \) is tested for \( i=1, \ldots, n \). If \( i \) is not used, \( a[k] \) is set to \( i \), used[\( i \)] is set to true, and “perm” calls itself recursively with parameter \( k+1 \). The meaning of “used” is to tell the positions downstream that item \( i \) is in use. After the call perm(\( k+1 \)), used[\( i \)] is reset to false. As the items are tested in increasing order at each position, obviously the order in which permutations are generated is lexicographic.

After “used” is initialized to false for all \( i \), the main program calls “perm” at position 1. If the parameter \( k \) becomes \( n+1 \), the current permutation in “a”, is output. When we measure the computation time with large \( n \), this output statement should be removed.

The algorithm is designed based on the simple idea of generating all \( n \)-ary strings, and removing non-eligible strings. The actual computation is slightly better than this, since the string “1 1”, for example, will cut further recursive calls. See the following for \( n=3 \).

\begin{verbatim}
1 1 1   1 3 1   2 2 1   3 1 1   3 3 1
1 1 2   1 3 2  ok 2 2 2   3 1 2  ok 3 3 2
1 1 3   1 3 3   2 2 3   3 1 3   3 3 3
1 2 1   2 1 1   2 3 1  ok 3 2 1  ok
1 2 2   2 1 2   2 3 2   3 2 2
1 2 3  ok 2 1 3  ok 2 3 3   3 2 3
\end{verbatim}
Meta Program
// This program generates a child program that generates
// permutations of size n. The value of n is given to
// this parent program. Then it generates a child program
// of 4-nested structure, which generates permutations."u" for "used".
#include <stdio.h>
FILE *out_file;
space(int k) {
    int i;
    for (i=1; i<=k; i++) fprintf(out_file, " ");
}
main() {
    int i, n;
    printf("Input n ");
    scanf("%d", &n);
    out_file=fopen("ff.c", "w");
    fprintf(out_file, "%s","main(){");
    fprintf(out_file, "int a[100], u[100];\n");
    fprintf(out_file, "int i;\n");
    for (i=1; i<=n; i++)
        fprintf(out_file, "int i%d;", i);
    fprintf(out_file, "\n");
    for (i=1; i<=n; i++) {
        space(4*i-4);
        fprintf(out_file, "for( i%d =1; i%d <= %d; i%d++)\{\n", i, i, n, i);
        space(4*i-2);
        fprintf(out_file, "if(u[i%d]==0)\{\n", i);
        fprintf(out_file, "{\n");
    }
    space(4*n-2);
    fprintf(out_file, "for( i=1; i<=n; i++)printf(" \%d
", a[i]);\}\n");
    fprintf(out_file, " printf("\n\n");\}\
");
    fprintf(out_file, "\n");
    for (i=n; i>=1; i--) {
        space(4*i-2);
        fprintf(out_file, "u[i%d]=0;\", i);
        fprintf(out_file, "\} // end if\);
        fprintf(out_file, "\n\n");
    }
Generated Program

```c
main(){int a[100], u[100];
    int i;
    int i1; int i2; int i3; int i4;
    for( i1 =1; i1 <= 4; i1++){
        if(u[i1]==0){u[i1]=1; a[1]=i1;
            for( i2 =1; i2 <= 4; i2++){
                if(u[i2]==0){u[i2]=1; a[2]=i2;
                    for( i3 =1; i3 <= 4; i3++){
                        if(u[i3]==0){u[i3]=1; a[3]=i3;
                            for( i4 =1; i4 <= 4; i4++){
                                if(u[i4]==0){u[i4]=1; a[4]=i4;
                                    for(i=1;i<=4;i++)printf("%d ,a[i]);
                                    printf(\n"");
                                    u[i4]=0; } // end if
                                } // end if
                            } // end if
                        } // end if
                    } // end if
                } // end if
            } // end if
        } // end if
    } // end if
}
```

To do time measurement, after “gcc meta.c”, run a script file “measure” in command line “time measure”, where “measure” is

```
a.out
gcc ff.dat
a.out
```
An example of testing program and tested program

Testing program
/* This program tests a sorting program by giving all permuted data sequences to it. The sorting program to be tested is in file "sort.c". The function sort(a,n) in it is supposed to sort array a of size n in non-decreasing order. This is in file"tester.c"
*/
#include <stdio.h>
#include "sort.c"
int i,j,k,m,n,c, kkk, flag;
int p[100]; int a[100]; int b[100];
int data[100];
copy(int a[], int b[]){
    int i;
    for(i=1;i<=n;i++)b[i]=a[i];
}
permute(int data[], int a[]){
    int i; int b[100];        // a[i] goes to position p[i] of b
        for(i=1;i<=n;i++)b[p[i]]=data[i];
        for(i=1;i<=n;i++)a[i]=b[i];
}
check(){int i, true;
    permute(data, a);  //data are permuted to a using p
    copy(a,b);         // a is copied to b
    sort(a,n);
    if(kkk==1)out();   // outputs are given on display
    else{
        true=1;
        for(i=1;i<=n-1;i++)true=true&&(a[i]<=a[i+1]);
        if(true==0){flag=1; printf("error
"); exit(1);} }
}
out(){
    int i;
    printf("data     "); for(i=1;i<=n;i++)printf("%3d ", data[i]);
    printf("\n");
    printf("permute  "); for(i=1;i<=n;i++)printf("%3d ", p[i]);
    printf("\n");
    printf("permutated "); for(i=1;i<=n;i++)printf("%3d ", b[i]);
    printf("\n");
    printf("sorted    "); for(i=1;i<=n;i++)printf("%3d ", a[i]);
    getchar();
    printf("\n");
}
swap(int i, int j) {
    int w;
    w = p[i]; p[i] = p[j]; p[j] = w;
}

int minimum(int k) {
    int i, j, temp;
    j = 0; temp = 99;
    for (i = k; i <= n; i++) {
        if ((p[i] < temp) && (p[i] > p[k - 1])) {
            temp = p[i]; j = i;
        }
    }
    return j;
}

reverse(int k) {
    int i;
    for (i = k; i <= (k + n - 1) / 2; i++) swap(i, n - i + k);
}

perm(int k) {
    int i, j;
    if (k <= n - 1) {
        for (i = k; i <= n; i++) {
            perm(k + 1);
            if (i < n) {
                reverse(k + 1);
                j = minimum(k + 1);
                swap(k, j); check(); // new permutation is checked
            }
        }
    }
}

main() {
    printf("input n "); scanf("%d", &n);
    getchar();
    printf("Do you output? 1 for yes, 0 for no ");
    scanf("%d", &kkk);
    for (i = 1; i <= n; i++) data[i] = random() % 1000;
    for (i = 1; i <= n; i++) p[i] = i;
    flag = 0;
    check();
    getchar();
    perm(1);
    if (flag == 0) printf("no errors ");
}
Tested program

// The insertion method. This is in file “sort.c”.
// This algorithm expands the sorted portion one by one
// by changing i as i=2, ..., n.
// temp is to keep the value of a[i].
// j scans the array portion a[1], ..., a[i-1] in reverse
// order to find the the position for temp. As long as
// a[j]>temp, it is shifted one place right.
#include <stdio.h>
#include <math.h>
sort(int a[], int n)
{
    int i, j, k, temp, t;
    for (i=2; i<=n; i++)
    {
        temp=a[i];
        for (j=i-1; j>=1; j--)
        {
            if (a[j]>temp) a[j+1]=a[j];
                else break;
        }
        a[j+1]=temp;
    }
A sample output for n=4

data 383 886 777 915
permute 1 2 3 4
permuted 383 886 777 915
sorted 383 777 886 915

data 383 886 777 915
permute 1 2 4 3
permuted 383 886 915 777
sorted 383 777 886 915

data 383 886 777 915
permute 1 3 2 4
permuted 383 777 886 915
sorted 383 777 886 915

data 383 886 777 915
permute 1 3 4 2
permuted 383 915 886 777
sorted 383 777 886 915

data 383 886 777 915
permute 1 4 2 3
permuted 383 777 886 915
sorted 383 777 886 915

data 383 886 777 915
permute 1 4 3 2
permuted 383 915 777 886
sorted 383 777 886 915

data 383 886 777 915
permute 2 1 3 4
permuted 886 383 777 915
sorted 383 777 886 915

data 383 886 777 915
permute 2 1 4 3
permuted 886 383 915 777
sorted 383 777 886 915

data 383 886 777 915
permute 2 3 1 4
permuted 777 383 886 915
sorted 383 777 886 915

data 383 886 777 915
permute 2 3 4 1
permuted 777 383 915 886
sorted 383 777 886 915

data 383 886 777 915
permute 2 4 1 3
permuted 777 383 915 886
sorted 383 777 886 915

data 383 886 777 915
permute 2 4 3 1
permuted 915 383 777 886
sorted 383 777 886 915
data     383 886 777 915
permute    3   1   2   4
permuted 886 777 383 915
sorted   383 777 886 915

data     383 886 777 915
permute    3   1   4   2
permuted 886 915 383 777
sorted   383 777 886 915

data     383 886 777 915
permute    3   2   1   4
permuted 777 886 383 915
sorted   383 777 886 915

data     383 886 777 915
permute    3   2   4   1
permuted 915 886 383 777
sorted   383 777 886 915

data     383 886 777 915
permute    3   4   1   2
permuted 886 915 777 383
sorted   383 777 886 915

data     383 886 777 915
permute    3   4   2   1
permuted 915 777 886 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   1   2   3
permuted 886 777 915 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   1   3   2
permuted 886 915 777 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   2   1   3
permuted 777 886 915 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   2   3   1
permuted 915 886 777 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   3   1   2
permuted 777 915 886 383
sorted   383 777 886 915

data     383 886 777 915
permute    4   3   2   1
permuted 915 777 886 383
sorted   383 777 886 915